A Workload for Evaluating Deep Packet Inspection Architectures

Michela Becchi, Mark Franklin and Patrick Crowley

IISWC '08
Context

- Pattern matching over *large* data-sets of *complex* regular expressions

- Application:
  - Networking: deep packet inspection
    - *Network Intrusion Detection and Prevention Systems*
    - Content based routing
    - Content based billing
    - Application level filtering
  - Others:
    - Bibliographic search

- Architecture:
  - Memory centric architectures (using cache)
In this paper

- Workload to evaluate memory-centric regular expression matching architectures
  - Synthetic rule-set generator
  - Traffic generator
  - Memory layout generator for NFA/DFA based designs

- Goal
  - Fair comparison between designs
  - Comprehensive tool
Background: handling multiple regex

Input text: …abcayxwknxKNZamkml…

Search patterns

- regex_1
- regex_2
- ...
- regex_N

NFA

DFA

FPGA designs

Memory-centric architectures

Linear processing time independent of number of patterns
Background: Finite Automata

RegEx: (1) $a^+bc$ (2) $bcd^+$ (3) $cde$

Text: $a \ b \ c \ d$

MEMORY BANDWIDTH:
# of state traversals per input character

Better for DFAs

MEMORY SIZE:
# states and transitions

Better for NFAs
Regular Expression Taxonomy

- **Exact-match strings**
  - *Fixed size patterns*
  - Properties:
    1. DFA size ≤ NFA size ≤ # chars in the pattern-set
    2. Multiple transitions to a state are on the same char
    3. Optimizations based on hashing schemes possible

    - [A. Aho and M. Corasick, CACM 1975]
    - [S. Dharmapurikar et al, ANCS 2005]
    - [N. Artan et al, INFOCOM 2007]
    - [Kumar et al, ICNP 2007]

  - Not expressive enough:
    - [R. Sommer and V. Paxson, CCS 2003]
    - [J. Newsome et al, Security and Privacy Symposium 2005]
    - [Y. Xie et al, SIGCOMM 2008]
Regular Expression Taxonomy (cont’d)

- **Character sets, single wildcards**
  - \([c_i\cdots c_k]\)
  - Properties:
    - Aho-Corasick and hashing schemes not directly applicable
    - *Exhaustive enumeration* of exact-match strings possible

- **Simple character repetitions**
  - \(c^+, c^*\)
  - Properties:
    - DFA size \(\sim\) number of characters in the pattern-set
    - Exhaustive enumeration of exact-match strings *not* possible
    - Hashing schemes not applicable
Regular Expression Taxonomy (cont’d)

- **Character sets and wildcards repetitions**
  - \.*\, \[^{c_i-c_j}\]*
  - Properties:
    - As for simple char repetitions
    - When compiling multiple regular expressions in the same DFA, DFA size can grow exponentially

  - Viable solutions:
    - NFA
    - Rule partitioning into multiple DFAs

  \begin{align*}
  \text{regex}_1 \quad & \text{regex}_2 \\
  \text{regex}_3 \quad & \text{regex}_4 \quad \ldots \quad \text{regex}_n
  \end{align*}

  - Each regex is compiled into an NFA, then converted to a DFA.
  - The result is a set of \(k\) concurrent DFAs, with \(k\) memory accesses per input character.
Regular Expression Taxonomy (cont’d)

- **Counting constraints**
  - \(c\{m,n\}\), \(sub-pattern\{m,n\}\)
  - \(.\{m,n\}\), \([^c_i-c_j]\{m,n\}\)

- Properties:
  - Exhaustive enumeration not feasible for large character ranges and large \(m,n\)
  - *Exponential* DFA size even on single regular expressions

- Viable solutions:
  - NFA
  - Hybrid-schemes using counters
In practice ...

- **As of November 2007**

| Data-set | # RegEx | \([c_1..c_n]\) | . | c+ | \(string^+\) | \([c_1..c_n]^*\) | \[^\n\r]*\) | .* | c\{n\} | \(string\{n\}\) | \([c_1..c_n]\{n\}\) | \{n\} |
|----------|---------|----------------|---|----|---------------|----------------|---------------|---|-----|----------------|----------------|----------------|------|
| Snort1   | 22      | 7              | 4 | 0  | 4             | 23             | 8             | 2 | 0   | 5              | 0              | 0              | 1    |
| Snort2   | 78      | 3              | 1 | 0  | 0             | 202            | 81            | 18 | 2   | 0              | 1              | 0              |      |
| Snort3   | 102     | 16             | 2 | 2  | 1             | 268            | 26            | 5  | 1   | 2              | 1              | 0              |      |
| Snort4   | 468     | 9              | 14| 3  | 7             | 113            | 468           | 38 | 0   | 7              | 11             | 3              |      |
| Bro0.8   | 226     | 1399           | 0 | 0  | 0             | 0              | 0             | 10 | 0   | 8              | 0              | 0              |      |
| Bro0.9   | 40      | 22             | 20| 0  | 6             | 1              | 0             | 0  | 0   | 10             | 0              | 0              |      |
| ClamAV   | 30411   | 0              | 0 | 0  | 0             | 0              | 1221          | 0  | 0   | 0              | 0              | 113             |      |

- **Over time**
  - Data-set size
  - Regular expression length
  - Number of (repeated) character ranges
  - Number of *dot-star*, \[^\n\r]* terms

are increasing!
Synthetic regex generation

- RegEx: alternation of *exact*- and *non-exact match sub-patterns*, according to frequency parameters
Traffic model

- **Goal:**
  - Generate synthetic traffic traces, rule-set dependent
  - Simulate different degrees of malicious activity

- **Observation:**
  - *Average/good traffic:*
    - limited to few low-depth states
    - high degree of *locality* (→ *fast path*)
  
  - *Bad traffic:*
    - partial matches → move to higher depth
    - low degree of *locality* (→ *slow path*)
      - non-repetitive input streams
      - ideally random walks in FA
Traffic model (cont’d)

- **Idea:**
  - $p_M$: probability of malicious traffic
  - FA based model: *given $p_M$ and set of active states, what is the next character in the input stream?*

- **Operation:**
  - *At each step:*
    1. Forward transition w/ $p_M$
    2. Random char w/ $(1 - p_M)$
  - *In case (1)*
    - **If** outgoing transitions exist
      - **Depth/active set size driven selection**
    - **else**
      - **Random char selection**
Memory encoding schemes

- **Note:**
  - NFA:
    - common prefixes collapsed
    - at most one epsilon tx/state
  - DFA:
    - default transition compression
      - [Kumar et al, SIGCOMM 2006]
      - [Becchi and Crowley, ANCS 2007]
        - At most $2N$ state traversal to process text of length $N$

- **Encoding schemes**
  - *Linear, bitmapped, indirect addressing*
  - **Affects**
    - Memory footprint
    - Cost of state traversal
Memory footprint

# DFAs: 1 – 2 – 2 – 14 – 24 - 32
## Experiments

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Traffic</strong></td>
<td></td>
</tr>
<tr>
<td>$p_M$</td>
<td>0.35, 0.55, 0.75, 0.95</td>
</tr>
<tr>
<td><strong>Cache</strong></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>4 KB, 16KB, 64KB, 256KB</td>
</tr>
<tr>
<td>line</td>
<td>64B</td>
</tr>
<tr>
<td>associativity</td>
<td>DM</td>
</tr>
<tr>
<td>hit latency</td>
<td>1 clock cycle</td>
</tr>
<tr>
<td>miss latency</td>
<td>30 clock cycles</td>
</tr>
<tr>
<td><strong>Memory layout</strong></td>
<td></td>
</tr>
<tr>
<td>encoding</td>
<td>linear, bitmapped, ind. addr 32-bit, ind. addr. 64-bit</td>
</tr>
</tbody>
</table>
State traversals/input char

# DFAs: 1 – 2 – 2 – 14 – 24 - 32

DFA
Rule-set clustering

NFA
$p_M$ affects active set size
Effect of state encoding

\[ p_M = 0.35 \]
Effect of cache size

**Indirect addressing, $p_M = 0.95$**

- **NFA**
  - 16KB cache size is sufficient

- **DFA**
  - on complex rule-set worse than NFA even with 256KB
Summary

- Proposal of workload to evaluate (memory-centric) regular expression matching architectures
  - Synthetic regular expression generator
  - Traffic trace generator
  - Memory layout generator
  - Cache simulator

- Model highlights:
  - Performance depends on
    - Rule-set size and complexity
    - NFA/DFA representation
    - Memory
    - Cache size

- On complex rule-sets, NFA can outperform DFAs
Thanks!

Questions?

REGEX tool download: http://regex.wustl.edu
Memory encoding scheme (cont’d)

- **Linear:**
  
  
<table>
<thead>
<tr>
<th>next addr</th>
<th>default/ε- tx</th>
</tr>
</thead>
<tbody>
<tr>
<td>c₁, next addr₁</td>
<td></td>
</tr>
<tr>
<td>c₂, next addr₂</td>
<td></td>
</tr>
<tr>
<td>cₖ, next addrₖ</td>
<td></td>
</tr>
</tbody>
</table>

  COST: Input dependent (linear traversal)

- **Bitmapped:**
  
<table>
<thead>
<tr>
<th>next addr</th>
<th>default/ε- tx</th>
</tr>
</thead>
<tbody>
<tr>
<td>next addr₁</td>
<td></td>
</tr>
<tr>
<td>next addrₖ</td>
<td></td>
</tr>
</tbody>
</table>

  COST:
  - Better for average traffic
  - Worse for “matching” traffic
Memory encoding scheme (cont’d)

- Indirect addressing:

  **state id:**
  \((c_1, c_2, \ldots c_k; \text{discriminator})\)

  *hash function*

  **next state id**
  
  **next state id\(_1\)**
  
  **next state id\(_k\)**

  **MEMORY**

  **default/ε- tx**

  **labeled tx**

  **COST**
  1 memory access/state traversal
Automata size

# DFAs: 1 - 2 - 2 - 14 - 24 - 32